A Parallel Distance-2 Graph Coloring Algorithm for Distributed Memory Computers*

Doruk Bozdağ[1], Umit Catalyurek[1], Assefaw H. Gebremedhin[2], Fredrik Manne[3], Erik G. Boman[4], and Füsun Özgüner[1]

[1] Ohio State University, USA, {bozdagd, ozguner}@ece.osu.edu, umit@bmi.osu.edu
[2] Old Dominion University, USA, assefaw@cs.odu.edu
[3] University of Bergen, Norway, Fredrik.Manne@ii.uib.no
[4] Sandia National Laboratories, USA, egboman@sandia.gov

Abstract. The distance-2 graph coloring problem aims at partitioning the vertex set of a graph into the fewest sets consisting of vertices pairwise at distance greater than two from each other. Application examples include numerical optimization and channel assignment. We present the first distributed-memory heuristic algorithm for this NP-hard problem. Parallel speedup is achieved through graph partitioning, speculative (iterative) coloring, and a BSP-like organization of computation. Experimental results show that the algorithm is scalable, and compares favorably with an alternative approach — solving the problem on a graph G by first constructing the square graph G^2 and then applying a parallel distance-1 coloring algorithm on G^2.

1 Introduction

An archetypal problem in the efficient computation of sparse Jacobian and Hessian matrices is the distance-2 (D2) vertex coloring problem in an appropriate graph [1]. D2 coloring also finds applications in channel assignment [2] and facility location problems [3]. It is closely related to a strong coloring of a hypergraph which in turn models problems that arise in the design of multifiber WDM networks [4]. The D2 coloring problem is known to be NP-hard [5].

In many parallel applications where a graph coloring is required, the graph is already distributed among processors. Under such circumstances, gathering the graph on one processor to perform the coloring may not be feasible due to memory constraints. Moreover, in some parallel applications the coloring needs to be performed repeatedly due to changes in the structure of the graph. Here, the coloring may take up a substantial amount of overall computation time unless a scalable algorithm is used.

A number of papers dealing with the design of efficient parallel distance-1 (D1) coloring algorithms have appeared [6–9]. For D2 coloring we are not aware of any work other than [10] where an algorithm for shared memory computers was presented.

In this paper, we present an efficient parallel D2 coloring algorithm suitable for distributed memory computers. The algorithm is an extension of the parallel D1 coloring

* This work was supported in part by NSF grants ACI-0203722, ACI-0203846, ANI-0330612, CCF-0342615, CNS-0426241, NIH NIBIB BISTI P20EB000591, Ohio Board of Regents BRTTC BRTT02-0003, Ohio Supercomputing Center PAS0052, and SNL Doc.No: 283793. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin company, for the U.S. DOE's National Nuclear Security Administration under contract DE-AC04-94AL85000.

algorithm presented in [6]. The latter is an iterative data parallel algorithm that proceeds in two-phased rounds. In the first phase, processors concurrently color the vertices assigned to them. Adjacent vertices colored in the same parallel step of this phase may result in inconsistencies. In the second phase, processors concurrently check the validity of the colors assigned to their respective vertices and identify a set of vertices that needs to be re-colored in the next round to resolve the detected inconsistencies. The algorithm terminates when every vertex has been colored correctly. To reduce communication frequency, the coloring phase is further decomposed into computation and communication sub-phases. During a computation sub-phase, a group of vertices, rather than a single vertex, is colored based on currently available color information. In a communication sub-phase processors exchange recent color information.

The key issue in extending this approach to the D2 coloring case is devising an efficient means of information exchange between processors hosting a pair of vertices that are two edges away from each other. We use a scheme in which the host processor of a vertex v is responsible for (i) coloring v, (ii) relaying color information to processors that store the D1 neighbors of v, and (iii) detecting inconsistencies that involve the D1 neighbors of v.

Our parallel D2 coloring algorithm has been implemented using MPI. Results from experiments performed on a 32-node PC cluster using a number of real-world as well as random graphs show that the algorithm is efficient and scalable. We have also compared our D2 coloring algorithm on a given graph G with the parallel D1 coloring algorithm from [6] applied to the square graph G^2. These results in general show that our algorithm scales better and uses less memory and storage.

In the sequel, we discuss preliminary concepts in Section 2; present our algorithm in Section 3; report experimental results in Section 4 and conclude in Section 5.

2 Distance-2 Graph and Hypergraph Coloring

Two distinct vertices in a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ are *distance-k* neighbors if the shortest path connecting them consists of at most k edges. A *distance-k coloring* of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a mapping $C : \mathcal{V} \to \{1, 2, \ldots, q\}$ such that $C(v) \neq C(w)$ whenever vertices v and w are distance-k neighbors. The associated optimization problem aims at minimizing q. A distance-k coloring of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is equivalent to a D1 coloring of the kth *power* graph $\mathcal{G}^k = (\mathcal{V}, \mathcal{F})$ where $(v, w) \in \mathcal{F}$ whenever vertices v and w are distance-k neighbors in $\mathcal{G}$. We denote the set of distance-k neighbors of vertex v by $N_k(v)$, and the set $N_k(v) \cup \{v\}$ by $N_k[v]$. For simplicity, we drop the subscript in the case where $k = 1$.

Let A be a symmetric matrix with nonzero diagonal elements and $\mathcal{G}_a(A) = (\mathcal{V}, \mathcal{E})$ be the *adjacency* graph of A, where $\mathcal{V}$ corresponds to the columns of A. As illustrated in the upper row of Figure 1, a partitioning of the columns of A into groups of structurally orthogonal columns is equivalent to a D2 coloring of $\mathcal{G}_a(A)$. (Two columns are structurally orthogonal if they do not have nonzero entries in the same row.) The right most subfigure in Figure 1 shows the equivalent D1 coloring in the square graph $\mathcal{G}_a^2$.

Now let A be non-symmetric. The *bipartite* graph of A is the graph $\mathcal{G}_b = (\mathcal{V}_1, \mathcal{V}_2, \mathcal{E})$ where $\mathcal{V}_1$ is the row vertex set, $\mathcal{V}_2$ is the column vertex set, and there exits an edge between row vertex r_i and column vertex c_j whenever $a_{ij} \neq 0$. As the lower row of

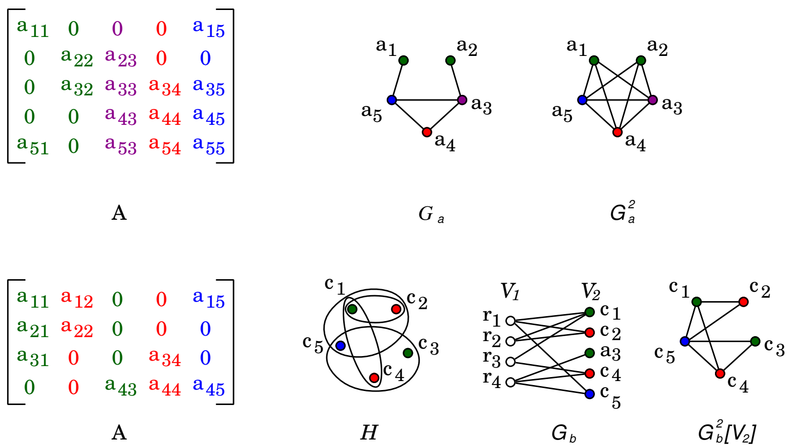


Fig. 1. Equivalence among structurally orthogonal column partition of A, D2 coloring of $\mathcal{G}(A)$ and D1 coloring of $\mathcal{G}^2(A)$. Top: symmetric case. Bottom: non-symmetric case (also shows equivalence with strong coloring of hypergraph H).

Figure 1 illustrates, a partitioning of the columns of A into groups of structurally orthogonal columns is equivalent to a partial D2 coloring of $\mathcal{G}_b(A)$ on $\mathcal{V}_2$. The right most subfigure shows the equivalent D1 coloring of $\mathcal{G}_b^2[\mathcal{V}_2]$, the subgraph of the square graph $\mathcal{G}_b^2$ induced by $\mathcal{V}_2$. The graphs $\mathcal{G}_a^2$ and $\mathcal{G}_b^2[\mathcal{V}_2]$ are also called *column intersection* graphs of A.

D2 coloring of a bipartite graph is also related to a variant of hypergraph coloring. A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ consists of a vertex set $\mathcal{V}$ and a collection $\mathcal{E}$ of subsets of $\mathcal{V}$ called *hyperedges*. A *strong hypergraph coloring* is a mapping $C : \mathcal{V} \to \{1, 2, \ldots, q\}$ such that $C(v) \neq C(v)$ whenever $\{v, w\} \subseteq e \in \mathcal{E}$. As Figure 1 illustrates, a strong coloring of a hypergraph is equivalent to a partial D2 coloring of its hyperedge-vertex incidence bipartite graph. For further discussion on the equivalence among matrix partitioning, D2 graph coloring and hypergraph coloring as well as their relationships to computation of Jacobians and Hessians, see [1].

3 Parallel Distance-2 Coloring

In this section we describe our new D2SC algorithm for a general graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Initially, the input graph is assumed to be distributed among p processors. The set V_i of vertices in the partition $\{V_1, \ldots, V_p\}$ of $\mathcal{V}$ is assigned to and colored by processor P_i; we say that P_i *owns* V_i. In addition, P_i stores the adjacency list of its vertices and the identities of the processors owning them. This classifies $\mathcal{V}$ into *interior* and *boundary* vertices. All D1 neighbors of an interior vertex are owned by the same processor as itself. A boundary vertex has at least one D1 neighbor owned by a different processor.

Clearly, any pair of interior vertices, that are assigned to different processors, can safely be colored concurrently. This is not true for a pair containing a boundary vertex. In particular, if such a pair is colored at the *same* parallel superstep, then the partners may receive the same color and result in a *conflict*. However, if we enforce that interior vertices be colored before or after boundary vertices, then a conflict can only occur for pairs of boundary vertices. Thus, the presented algorithm is concerned with parallel coloring of boundary vertices.

The main idea in our algorithm is to color boundary vertices concurrently in a speculative manner and then detect and rectify conflicts that may have arisen. The algorithm is iterative—it proceeds in *rounds*. Each round consists of a *tentative coloring* and a *conflict detection* phase. Both of these phases are performed in parallel. The latter phase detects conflicts in a current coloring and accumulates a list of vertices to be recolored in the next round. Given a pair of vertices involved in a conflict, only one of them needs to be recolored to resolve the conflict; the choice is done randomly. The algorithm terminates when there are no more vertices to be colored. The high-level structure of the algorithm is outlined in Algorithm 1.

Algorithm 1 An iterative parallel distance-2 coloring algorithm

---

procedure PARALLELCOLORING($\mathcal{G} = (\mathcal{V}, \mathcal{E}), s$)
  Initial data distribution: $\mathcal{G}$ is divided into p subgraphs $G_1 = (V_1, E_1), \ldots, G_p = (V_p, E_p)$ where $V_1, \ldots, V_p$ is a partition of the set $\mathcal{V}$ and $E_i = \{(v, w) : v \in V_i, (v, w) \in \mathcal{E}\}$. Processor P_i owns the vertex set V_i, and stores the edge set E_i and the ID's of the processors owning the other endpoints of E_i.
  on each processor P_i, $i \in P = \{1, \ldots, p\}$
    Color interior vertices in V_i
    $U_i \leftarrow$ boundary vertices in V_i ▷ U_i is to be iteratively colored by P_i
    while $\exists j \in P$, $U_j \neq \emptyset$ **do**
      $W_i \leftarrow$ COLOR(G_i, U_i, s)   ▷ W_i is examined for conflicts by P_i
      $U_i \leftarrow$ DETECTCONFLICTS(G_i, W_i)

Even if a processor P_i currently has no vertices to color ($U_i = \emptyset$), it could still be active since other processors may require color information from P_i. Furthermore, P_i may participate in detecting conflicts on other processors. For every path v, w, x, the host processor for vertex w is responsible for detecting D2 conflicts that involve vertices v and x as well as D1 conflicts involving w and its adjacent vertices $N(w)$. The set W_i in Algorithm 1 contains the current set of vertices residing on processor P_i that will be examined for detecting these conflicts. The set W_i includes both 'middle' vertices as well as vertices from U_i. The discussion of the routines COLOR and DETECTCONFLICTS in Sections 3.1 and 3.2 will clarify these points.

3.1 The Tentative Coloring Phase

The tentative coloring phase is organized as a sequence of *supersteps*. In each superstep, each processor colors s vertices sequentially, and sends the colors of these vertices to processors owning their D1 neighbors. To perform the coloring, a processor first gathers information from other processors to build a (partial) list of forbidden colors for each of its boundary vertices scheduled to be colored in the current superstep. Such a list for a vertex v consists of the colors used by its already colored D2 neighbors. The colors of the off-processor vertices in $N(v)$ colored in previous supersteps are easily available since the host processor of v has already received and stored them. We refer to such colors as *local*. However, the colors used by vertices exactly two edges away from v may have to be obtained from another processor.

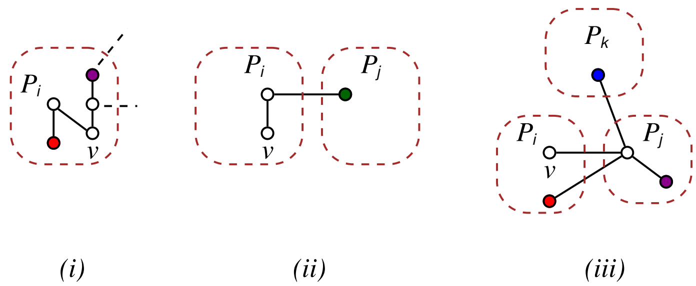


Fig. 2. Distribution scenarios of the distance-2 neighbors of vertex v across processors.

Figure 2 illustrates the three scenarios in which the vertices on a path v, w, x may be distributed among processors. Case *(i)* corresponds to the situation where both w and x are owned by P_i. Case *(ii)* shows the situation where w is owned by P_i and x is owned by P_j, $j \neq i$. In these two cases, the color of w is local to P_i. Case *(iii)* shows the situation where w is owned by P_j, and vertices v and x do not have a common D1 neighbor owned by P_i. Vertex x may be owned by any one of the three processors P_i, P_j, or P_k, $i \neq j \neq k$. In the third case, the color of x is not local to P_i and needs to be relayed through P_j which is capable of detecting the situation. In particular, P_j builds and sends a list of forbidden colors for each vertex owned by P_i that P_i cannot access directly. Since P_j does not know the internal structure of the vertices in P_i, it includes the color of every x in the list of forbidden colors for v for each path v, w, x where w is owned by P_j.

At the beginning of the algorithm, each processor sends a coloring-schedule of its boundary vertices to neighboring processors. In this way, each processor will know the D2 color information it needs to send in each superstep. Note that it is only necessary to send information regarding D1 neighbors of a vertex owned by another processor. Each processor then computes a list X_i of vertices on neighboring processors for which it must supply color information. With the knowledge of X_i, processor P_i can now be "pro-active" in building and sending lists of relevant color information. When a processor receives the partial lists of forbidden colors from all of its neighboring processors, it merges these lists with local color information to determine a complete list of forbidden colors for its vertices scheduled to be colored in the current superstep. Using this information, a processor then speculatively colors these vertices and sends the new color information to processors owning D1 neighbors.

In addition to coloring vertices in the current set U_i, a processor also builds and returns a list W_i of vertices that it needs to examine in the conflict detection phase. Two vertices are involved in a conflict only if they are colored in the same superstep. Thus W_i consists of (i) every vertex that has at least two neighbors on different processors that are colored in the same superstep, and (ii) every vertex v in U_i that has at least one neighbor on a processor P_j, $j \neq i$, colored in the same superstep as v. The tentative coloring routine is outlined with more details in Algorithm 2.

The set W_i is efficiently determined in the following manner. The vertices in $X_i \cup U_i$ are traversed a superstep at a time. For each superstep, first, each vertex in U_i and its neighboring boundary vertices are *marked*. Then for each vertex $v \in X_i$ the vertices in $N(v)$ owned by processor P_i are marked. If this causes some vertex to be marked twice in the same superstep, then the vertex is added to W_i. The combined sequential work carried out by P_i and its neighboring processors to perform the coloring of U_i is $O(\Sigma_{v \in U_i} |h(v)|)$ where $h(v)$ is the graph induced by the edges incident on $N[v]$. Summing over all processors, the total work involved in coloring the vertices in $U = \cup U_i$ is $O(\Sigma_{v \in U} |h(v)|)$ which is equivalent to the complexity of a sequential algorithm.

Algorithm 2 Speculative coloring.

function COLOR(G_i, U_i, s)
  Partition U_i into ℓ_i subsets $U_{i,1}, U_{i,2}, \ldots, U_{i,\ell_i}$, each of size s, and send the schedule to relevant processors
  $X_i \leftarrow \bigcup_{j,k} U^i_{j,k}$ ▷ $U^i_{j,k}$: vertices received by P_i to be colored by P_j in step k
  for each $v \in U_i \cup X_i$ **do**
    $C(v) \leftarrow 0$ ▷ (re)initialize colors
  $W_i \leftarrow \emptyset$ ▷ W_i is used for detecting conflicts
  for each $v \in V_i$ s.t v has at least two neighbors in $X_i \cup U_i$ on different processors, both colored in the same superstep **do**
    $W_i \leftarrow W_i \cup \{v\}$
  for each $v \in U_i$ s.t v has at least one neighbor in X_i that is colored in the same superstep as v **do**
    $W_i \leftarrow W_i \cup \{v\}$
  for $k_i \leftarrow 1$ **to** ℓ_i **do** ▷ each k_i corresponds to a superstep
    for each neighboring P_j where $k_j < \ell_j$ **do** ▷ P_j is not in its last superstep
      Build and send lists of forbidden colors to P_j for relevant vertices in U_{j,k_j+1}
    Receive and merge lists of forbidden colors for relevant vertices in U_{i,k_i}
    Update lists of forbidden colors with local color information
    for each $v \in U_{i,k_i}$ **do**
      $C(v) \leftarrow c$ s.t. $c \neq 0$ is the smallest permissible color for v
    Send colors of relevant vertices in U_{i,k_i} to neighboring processors
  while $\exists j \in P$, s.t. P_j is a neighbor of P_i and $k_j \leq l_j$ **do**
    Receive color information for superstep k_j from P_j
    if $k_j < l_j$ **then**
      Build and send list of forbidden colors to P_j for relevant vertices in U_{j,k_j+1}
  return W_i

For each $v \in U_i$, every neighboring processor sends the union of the colors used by vertices at exactly 2 edges from v, while the color of v is sent back to every processor that stores a neighbor of v. The only time that the color of v might be sent more than once to P_i is if there exists a triangle v, w, x such that w is owned by P_j and x by P_k, where i, j, and k are all different. Then P_j adds the color of x to the list of forbidden colors to be sent to P_i, while P_k adds the color of w to the list it sends to P_i. Thus, the overall size of communicated data is bounded by $\Sigma_{v \in \cup U_i} |h(v)|$.

The discussion above implies that a partitioning of $\mathcal{G}$ among processors where the number of boundary vertices is small relative to the number of interior vertices on each processor is highly desirable as it reduces communication cost.

3.2 The Conflict Detection Phase

A conflict involving a pair of adjacent vertices is detected by both processors owning these vertices. A conflict involving a pair of vertices exactly two edges apart is detected by the processor owning the middle vertex. To resolve a conflict, one of the involved vertices is randomly chosen to be recolored in the next round. Algorithm 3 outlines the parallel conflict detection phase DETECTCONFLICTS executed on each processor P_i. This routine returns a set of vertices to be colored in the next round by P_i.

Algorithm 3 Conflict Detection

1: **function** DETECTCONFLICTS(G_i, W_i)
2: $R_{i,j} \leftarrow \emptyset$ for each $j \in P$ ▷ $R_{i,j}$ is a set of vertices P_i notifies P_j to recolor
3: **for each** vertex $w \in W_i$ **do**
4: $seen[C(w)] \leftarrow w$
5: $where[C(w)] \leftarrow w$
6: **for each** $x \in N(w)$ **do**
7: **if** $seen[C(x)] = w$ **then**
8: $v \leftarrow where[C(x)]$
9: **if** $r(v) \leq r(x)$ **then** ▷ $r(x)$ is a random number associated with x
10: $R_{i,I(x)} \leftarrow R_{i,I(x)} \cup \{x\}$ ▷ $I(u)$ is ID of processor owning u
11: **else**
12: $R_{i,I(v)} \leftarrow R_{i,I(v)} \cup \{v\}$
13: $where[C(x)] \leftarrow x$
14: **else**
15: $seen[C(x)] \leftarrow w$
16: $where[C(x)] \leftarrow w$
17: **for each** $j \neq i \in P$ **do**
18: send $R_{i,j}$ to processor P_j
19: **for each** $j \neq i \in P$ **do**
20: receive $R_{j,i}$ from processor P_j
21: $R_{i,i} \leftarrow R_{i,i} \cup R_{j,i}$
22: **return** $R_{i,i}$

Each processor P_i accumulates and sends a list $R_{i,j}$ of vertices to be recolored by each P_j in the next round. P_i is responsible for recoloring vertices in $R_{i,i}$ and therefore adds received notifications $R_{j,i}$ from each neighboring processor P_j to $R_{i,i}$.

To efficiently determine the subset of W_i that needs to be recolored, we use two color-indexed tables $seen[]$ and $where[]$. The assignment $seen[c] = w$ is effected if at least one vertex in $N[w]$ of color c has been processed. The entry $where[c]$ stores the vertex with the lowest random value among these. Initially both $seen[C(w)]$ and $where[C(w)]$ are set to w. This ensures that any conflict involving w and a vertex in $N(w)$ will be discovered. For subsequent neighbors of w, a check on whether their colors have already been seen is first done. If positive, the vertex that needs to be recolored is determined based on its random value and the table $where$ is updated accordingly.

Note that in Line 6, it is sufficient to only check for conflicts using vertices that are both in $N(w)$ and in either U_i or X_i. However, determining which vertices in $N(w)$ this applies to takes more time than testing for a conflict. Also, it is not necessary to notify a neighboring processor on the detection of a conflict involving adjacent vertices as the conflict will also be discovered by the other processor.

4 Experimental Results

We carried out experiments on a 32-node PC cluster equipped with dual 2.4 GHz Intel P4 Xeon CPUs with 4 GB of memory. The nodes are interconnected via a switched

10Gbps Infiniband network. Our test set consists of 21 graphs from molecular dynamics and finite element applications [11, 7, 12, 13]. We report average results for each class of graphs, instead of individual graphs. Each result is in turn an average of 5 runs.

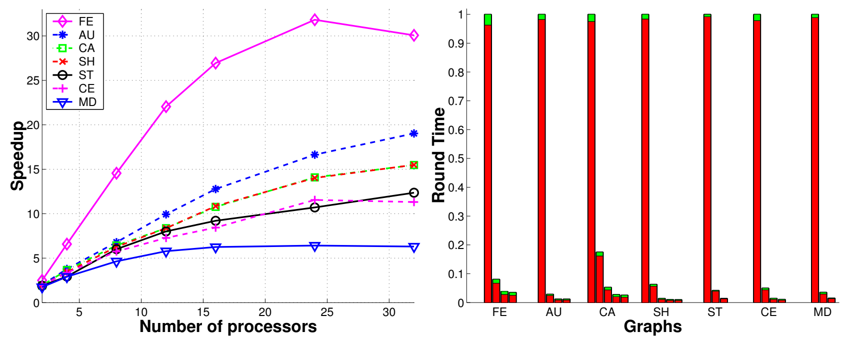
app	name	$\lvert V\rvert$	$\lvert E\rvert$	Degree max	Degree avg	D1 time	D1 col	D2 time (norm.)	D2 col	D1 on $\mathcal{G}^2$ (norm.) $\times\lvert E\rvert$	D1 on $\mathcal{G}^2$ (norm.) conv. time	D1 on $\mathcal{G}^2$ (norm.) color. time
MD	popc-br-4	24,916	255,047	43	20	3.3	21	20.6	75	4.7	33.0	4.8
	er-gre-4	36,573	451,355	42	25	5.5	19	23.2	66	5.0	34.6	5.1
	apoa1-4	92,224	1,131,436	43	25	16.5	20	18.1	73	5.0	28.5	4.3
FE	144	144,649	1,074,393	26	15	44.3	12	20.5	41	4.8	25.8	4.0
	598a	110,971	741,934	26	13	35.0	11	20.9	38	4.7	28.3	4.0
	auto	448,695	3,314,611	37	15	248.9	13	16.1	42	4.9	16.6	4.5
CA	bmw3_2	227,362	5,530,634	335	49	53.3	48	42.7	336	3.2	35.7	3.0
	bmw7st1	141,347	3,599,160	434	51	34.4	54	43.8	435	3.3	35.6	3.0
	inline1	503,712	18,156,315	842	72	179.5	51	70.5	843	7.0	63.8	6.2
ST	pwtk	217,918	5,708,253	179	52	50.7	48	45.5	180	2.9	34.8	2.7
	nasasrb	54,870	1,311,227	275	48	11.7	41	42.8	276	3.2	35.5	3.2
	ct20stif	52,329	1,323,067	206	51	12.5	49	46.0	210	3.8	37.7	3.5
AU	hood	220,542	5,273,947	76	48	58.5	42	35.8	103	3.2	29.2	2.7
	ldoor	952,203	22,785,136	76	48	249.7	42	35.8	112	3.2	29.0	2.7
	msdoor	415,863	9,912,536	76	48	106.2	42	36.9	105	3.2	29.8	2.7
CE	pkustk10	80,676	2,114,154	89	52	20.1	42	43.6	126	2.9	33.0	2.7
	pkustk11	87,804	2,565,054	131	58	23.7	66	57.6	198	4.2	45.6	3.8
	pkustk13	94,893	3,260,967	299	69	29.2	57	72.5	303	6.0	62.2	6.0
SH	shipsec1	140,874	3,836,265	101	54	34.9	48	48.5	126	3.1	37.7	8.2
	shipsec5	179,860	4,966,618	125	55	46.4	50	48.5	140	3.2	37.2	3.0
	shipsec8	114,919	3,269,240	131	57	29.1	54	52.9	150	3.5	41.2	3.4

Table 1. Structural properties of test graphs (left). Performance results (right). Sources: MD [13]; FE [7]; CA, SH [11]; ST, AU, CE [12].

The left half of Table 1 displays the structural properties of the test graphs classified according to application area. The first part of the right half lists the number of colors and the runtime in milliseconds used by a sequential D1 coloring algorithm. The second part shows timings for two different ways of (sequentially) performing a D2 coloring on $\mathcal{G}$. The time spent on constructing $\mathcal{G}^2$ from $\mathcal{G}$ is given under column *conv. time*. The reported times have been normalized with respect to the corresponding time required for performing a D1 coloring. We also list the ratio of the number of edges in $\mathcal{G}^2$ to that $\mathcal{G}$, to show the relative increase in storage requirement.

As one can see, $\mathcal{G}^2$ requires a factor of nearly 3 to 7 more storage than $\mathcal{G}$. D2 coloring on $\mathcal{G}$ is in most cases slightly slower than constructing and then D1 coloring $\mathcal{G}^2$. We believe this is due to the fact that a D1 coloring on $\mathcal{G}^2$ accesses memory more sequentially in comparison with a D2 coloring on $\mathcal{G}$.

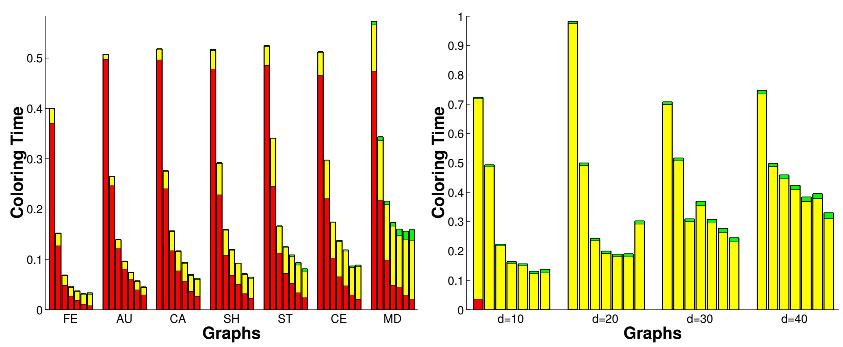
Figure 3(a) shows the obtained speedups for the different graph classes while keeping the superstep size fixed at 100. For most graph classes, reasonable speedup is obtained as the number of processors is increased. We have also conducted experiments to investigate the impact of superstep size. We found that with the exception of extreme values, superstep size does not significantly influence speedup.



(a) (b)

Fig. 3. (a) Speedup while varying number of processors for $s = 100$. (b) Breakdown of execution time into rounds and coloring and conflict detection phases for $p = 16$. Each bar is divided into time spent on coloring (bottom) and time spent on conflict detection (top).

We observed that the number of conflicts increases with increasing number of processors and superstep size. Still, it stays fairly low and does not exceed 10% of the number of vertices with the exception of MD graphs with up to 32 processors for $s = 100$. The number of rounds the algorithm has to iterate was observed to be consistently low, increasing only slowly with superstep size and number of processors. This is due to the fact that the number of initial conflicts drops rapidly between successive rounds. To further show how the time within each round is spent we present Figure 3(b). The figure shows the time spent on coloring boundary vertices and conflict detection in each round for 16 processors. All timings are normalized with respect to the time spent in the first round, excluding the time spent on coloring interior vertices.



(a) Graphs in Table 1 (b) Random graphs

Fig. 4. Breakdown of execution time into time spent on coloring internal vertices (bottom), coloring boundary vertices (middle), and conflict detection (top). For each graph class, timings for $p = 2, 4, 8, 12, 16, 24, 32$ are reported.

Figure 4(a) shows how the total time is divided into time spent on coloring interior vertices, coloring boundary vertices, and conflict detection. All timings are normalized with respect to the sequential coloring time. As the number of processors increases, the time spent on coloring boundary vertices does not change much while the time spent on coloring interior vertices decreases almost linearly. This should be seen in light of the fact that the number of boundary vertices increases as more processors are applied whereas the coloring of interior vertices does not involve any communication.

To investigate scalability on boundary vertices, we performed experiments on random graphs. For a random graph almost every vertex becomes a boundary vertex regardless of how the graph is partitioned. We generated random graphs with 100,000 vertices and with average degrees of 10, 20, 30, and 40. Figure 4(b) shows that the algorithm scales fairly well and almost all the time is spent on coloring boundary vertices.

We have also evaluated D1 coloring on $\mathcal{G}^2$ and experimental results (omitted due to space constraints) indicate that the D1 coloring on $\mathcal{G}^2$ approach is less scalable and requires more storage than the D2 coloring on $\mathcal{G}$ approach. However, since our timings for the $\mathcal{G}^2$ approach did not include the time to construct $\mathcal{G}^2$ our observation is not conclusive. We intend to investigate this issue in a future work.

5 Conclusion

We have presented an efficient parallel distance-2 coloring algorithm suitable for distributed memory computers and experimentally demonstrated its scalability. In a future work we plan to adapt the presented algorithm to solve the closely related strong hypergraph coloring problem. This brings up the open problem of finding a suitable partition of the vertices and edges of a hypergraph.

References

1. Gebremedhin, A.H., Manne, F., Pothen, A.: What color is your jacobian? Graph coloring for computing derivatives. SIAM Rev. (2005) To appear.
2. Krumke, S., Marathe, M., Ravi, S.: Models and approximation algorithms for channel assignment in radio networks. Wireless Networks **7** (2001) 575 – 584
3. Vazirani, V.V.: Approximation Algorithms. Springer (2001)
4. Ferreira, A., Pérennes, S., Richa, A.W., Rivano, H., Stier, N.: Models, complexity and algorithms for the design of multi-fiber wdm networks. Telecommunication Systems **24** (2003) 123 – 138
5. McCormick, S.T.: Optimal approximation of sparse hessians and its equivalence to a graph coloring problem. Math. Programming **26** (1983) 153 – 171
6. Boman, E.G., Bozdağ, D., Catalyurek, U., Gebremedhin, A.H., Manne, F.: A scalable parallel graph coloring algorithm for distributed memory computers. (EuroPar 2005, to appear)
7. Gebremedhin, A.H., Manne, F.: Scalable parallel graph coloring algorithms. Concurrency: Practice and Experience **12** (2000) 1131–1146
8. Gebremedhin, A.H., Manne, F., Woods, T.: Speeding up parallel graph coloring. In: proceedings of Para 2004, Lecture Notes in Computer Science, Springer (2004)
9. Jones, M.T., Plassmann, P.: A parallel graph coloring heuristic. SIAM J. Sci. Comput. **14** (1993) 654–669
10. Gebremedhin, A.H., Manne, F., Pothen, A.: Parallel distance-k coloring algorithms for numerical optimization. In: proceedings of Euro-Par 2002. Volume 2400., Lecture Notes in Computer Science, Springer (2002) 912–921
11. : (Test data from the parasol project) http://www.parallab.uib.no/projects/parasol/data/.
12. : (University of florida matrix collection) http://www.cise.ufl.edu/research/sparse/matrices/.
13. Strout, M.M., Hovland, P.D.: Metrics and models for reordering transformations. In: proceedings of MSP 2004. (2004) 23–34